ReactJS

m2iformation.fr

Sommaire

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction

Créer une application

Pour créer une application React, il faut déjà s'assurer que l'on possède Node.js. Pour ce faire, il est possible d'utiliser la commande :

```
node -v
```

Une fois ceci fait, pour créer une application react, la commande à lancer est :

```
npx create-react-app nom_d_application
# OU
npm create vite@latest nom_d_application --template react
```

Le format .jsx

Le fichier `App.jsx` (ou `App.tsx`) nous sert de base pour le placement de nos futurs composants, qui à terme seront porteurs de l'extension `.js`. Dans l'univers React, il existe cependant la capacité de retourner de l'HTML mêlé à du Typescript / Javascript. Pour cela, nous avons besoin d'un type de donnée mêlant les deux. Ce type de donnée est en réalité du **JSX / TSX** et il est possible de le stocker dans des variables, de le retourner en fin de fonction, etc...

Première Application

Les Composants

L'avantage de travailler avec React est la capacité de réaliser des **composants**. Les composants sont des éléments React retournant une structure HTML via un retour de type **JSX**. Pour placer des composants à un endroit précis de notre application, il suffit par la suite de nous servir d'une syntaxe du genre:

- Pour un composant simple:

```
<NomComposant />
```

- Pour un composant plus générique:

```
<NomComposant> ... </NomComposant>
```

Ajouter des valeurs métier dans le template

Si l'on souhaite placer des variables ou des expressions Javascript au sein de notre structure HTML, on peut le faire via l'utilisation d'une syntaxe de l'ordre de:

```
let maVariable = "Blabla";

return (
  <div>
    La valeur de la variable est {maVariable}
  </div>
)
```

Styliser notre application

Si l'on veut désormais ajouter les fichiers de style `.css` (ou autre extension), il est possible de les importer de deux façons:

- Pour rendre les fichiers de styles globaux:

```
import 'chemin/vers/fichier.css'
```

- Pour créer des classes disponibles dans un composant uniquement:

```
import classes from 'chemin/vers/fichier.module.css'
```

Si l'on se sert de modules de styles encapsulés:

```
<div class={classes.maClasse}> ... </div>
```

Incorporer des éléments dans notre JSX

Si l'on souhaite avoir des images dans notre application React, il va falloir:

- Soit utiliser des images placées dans le dossier public (ou autre sous-dossier) de l'application et utiliser un lien de type absolu:

```
<img src="/images/monImage.png" alt="" />
```

- Soit placer des valeurs dans les attributs `src` correspondant à un import de l'image en tant qu'élément de code:

```
<img src={monImage} alt="" />
```

Passer des informations vers un composant

Pour faire passer nos variables ou des résultats d'expressions Javascript en tant que valeurs d'un composant, il est possible d'avoir recours aux `props`:

```
<MonComposant paramA="valueA" paramB="valueB">
```

Au sein du composant, il est ensuite possible de les récupérer de la sorte:

```
const MonComposant = (props) => {
  let paramA = props.paramA
  let paramB = props.paramB
}
```

Réagir à l'utilisateur

Pour pouvoir permettre à l'utilisateur d'interagir avec notre application, on va passer par les évènements. Dans React, la majorité des balises disposent d'évènements de type `onTypeEvenement` tel que:

```
<button onClick={...}> Click Me! </button>
```

On peut alors en valeur de l'attribut placer soit la référence d'une fonction Javascript, soit une fonction fléchée de sorte à fixer les paramètres (par défaut, seul l'évènement est envoyé en paramètre à une référence):

```
<button onClick={() => console.log('Hello World')}> Click Me! </button>
```

Traiter l'évènement

Dans le cas où l'on aurait besoin d'avoir accès à l'évènement levé par le navigateur pour tel type d'interaction utilisateur, il est possible de le demander:

```
const gestionEvent = (event, otherParam) => {
  // Code
}
```

```
<button onClick={gestionEvent}> Click Me! </button>
<button onClick={(e) => gestionEvent(e, autreParam)}> Click Me! </button>
```

Déclencher chez le parent (1/2)

Pour pouvoir permettre à un composant enfant de déclencher, via un évènement, du comportement chez le parent, on va encore une fois passer par les props. Cette fois-ci, on va cependant passer en valeur une référence de fonction:

```
const ParentComponent = () => {
  const myFunction = () => {
    // ...
  }

  return (
    <div>
      <ChildComponent paramA={myFunction} />
    </div>
  )
}
```

Déclencher chez le parent (2/2)

Au niveau de l'enfant, il ne reste plus qu'à faire appel à la méthode issue des props. En fonction de si l'on a besoin de l'évènement, on aura plusieurs syntaxes donc:

- Avec évènement:

```
<button onClick={props.paramA}> Click Me! </button>
<button onClick={(e) => props.paramA(e, autreParam)}> Click Me! </button>
```

- Sans évènement:

```
<button onClick={() => props.paramA(autreParam)}> Click Me! </button>
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Le State en React

Qu'est-ce que le State ?

Dans une application, il y a en permanence des changements, tels que la création d'objets, leur modification, leur destruction, etc... Tout ceci mène à une modification de l'état de l'application. Pour pouvoir gérer cet état, il faut d'abord créer la capacité de faire des composants React dits **Stateful**, par la création, non par de fonctions, mais de classes.

Depuis, les choses ont bien changé et il est désormais possible de gérer l'état de l'application dans les composants faits à partir de fonctions (les originels **Stateless**) grâce aux **Hooks**.

Les classes Component / PureComponent

Lorsque l'on veut créer un composant se servant du state, il convient donc de créer de base une classe étendant l'une des deux classes offrant la gestion du state:

```
import { PureComponent } from 'react'

class MyComponent extends PureComponent {
  constructor(props) {
    super(props)
    this.state = {
      // Ici on mettra l'état de l'application
    }

    render() {
      return (
        // Ici sera le template HTML
      )
    }
  }
}
```

Remplacer le state

Pour par la suite modifier le state, il nous faudra, au sein de notre code Javascript, faire appel à une méthode héritée : `setState()`. Cette méthode prendra en paramètre un objet dont les attributs doivent être les mêmes que les variables de state fixées au départ. Seules ces variables se verront modifiées et seuls les éléments de template usant ces valeurs se verront re-rendus dans le HTML (ce au moyen du principe de Shadow DOM).

```
myMethod(numberA, numberB) {
  this.setState({sum: numberA + numberB})
}
```

Editer le state

Dans le cas où l'on aurait besoin d'avoir accès aux valeurs précédentes du state afin de les modifier (et non les remplacer), on peut utiliser d'autres syntaxes dans notre méthode `setState()`:

```
myMethod(numberA, numberB) {
  this.setState(previousState => {
    ...previousState,
    sum: previousState.sum + numberA + numberB
  })
}
```

Le Cycle de Vie

- `componentDidMount()`: cette méthode sera appelée suite à une demande de rendu dans le HTML du composant. C'est dans cette méthode que nous placerons idéalement les appels API permettant l'initialisation des données de notre composant.
- `componentDidUpdate()`: cette méthode sera appelée lorsque notre composant se voit être re-rendu. Par exemple, si celui-ci utilise un state particulier, et que ce state se voit modifié, alors le composant se verra modifié et cette méthode sera appelée par React.

Le Cycle de Vie

- `componentDidCatch()` : cette méthode sert lorsque notre composant se voit être présenté une erreur et que l'on veut la traiter. Il est possible de catcher l'erreur, puis de traiter la récupération d'une erreur au niveau du composant dans cette méthode.
- `componentWillUnmount()` : cette méthode se voit appelée dans le cas où notre composant est amené à disparaitre. Par exemple, si un composant doit disparaitre suite à un changement de page, ou le changement d'un rendu conditionnel, alors on peut appliquer une logique métier à ce moment précis.

Les Hooks

Désormais, il est possible d'avoir un fonctionnement similaire et d'avoir un state dans les composants de type fonction. Pour cela, nous avons recours aux **Hooks** qui sont des fonctions disponibles en React. Il en existe plusieurs, et chaque hook a son fonctionnement propre. Plus légers, il reste cependant l'impossibilité de se brancher aux moments du cycle de vie des composants via des composants fonctionnels, donc attention à ne pas choisir le mauvais type de composant pour un objectif précis.

useState()

Le premier hook, et sans doute celui qui est le plus utilisé, est `useState()`. Son objectif est la capacité de gérer et de modifier un état tout comme le permettait l'attribut `state` et la méthode `setState()` des classes vues précédemment. Le state peut être aussi complexe qu'on le désire, mais attention à son côté immutable car c'est au moment d'un changement de valeur / référence de la constante que le re-rendu se voit provoqué:

```
import { useState } from 'react'

const [maVariableDeState, setMaVariableDeState] = useState(valeurInitiale)
```

Changer la valeur

Comme via l'utilisation de `setState()` dans la version classe, il existe deux façons de changer la valeur de l'état:

- Si l'on veut remplacer la valeur:

```
setMaVariableDeState(nouvelleValeur)
```

- Si l'on veut éditer la valeur:

```
setMaVariableDeState(ancienEtat => ancienEtat + nouvelleValeur)
```

```
setMaVariableDeState(ancienEtatArray => [...ancienEtat + nouvelleValeur])
```

```
setMaVariableDeState(ancienEtatObj => {...ancienEtatObj, cle: nouvelleValeur})
```

useEffect()

Si l'on veut désormais provoquer des instructions au lancement ou à chaque re-rendu d'un composant, il est possible d'utiliser le hook useEffect():

```
useEffect(() => {
  /* La fonction au sein de useEffect() sera déclenchée à chaque rendu du
  composant ou à chaque modification d'une des dépendances placées dans le
  tableau offert en second argument au hook */

  return () => {
     /* La fonction retournée par useEffect() sera déclenchée à chaque
     suppression (re-rendu également) de notre composant */
    }
}, [...dependances])
```

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Le Rendu Conditionnel

Retour conditionnel

Un composant React n'étant au final qu'un élément retournant du JSX, on peut faire un bloc de type IF...ELSE IF...ELSE... en amont du retour et avoir ainsi plusieurs retours possibles:

```
if (maVariable === maValeur) {
  return ( <div> Elements si vrai </div> )
} else {
  return ( <div> Elements si faux </div> )
}
```

Malheureusement, cette méthode va nous forcer à écrire plusieurs fois les éléments en commun entre les X rendus conditionnels de notre composant.

Au sein du retour

Les composants utilisent du **JSX** pour mêler le Javascript à l'HTML. Cette particularité de React nous offre la capacité de réaliser des ternaires retournant non pas des valeurs Javascript mais directement du JSX:

```
{evaluationBooleenne ? <p>Vrai</p> : <p>Faux</p>}
```

Dans le cas où l'on ne souhaiterait pas avoir d'affichage en cas de valeur fausse, on peut utiliser le principe de l'opérateur logique **AND**:

```
{evaluationBooleenne && <p>Vrai</p>}
```

Les Fragments

Si l'on a envie de retourner du JSX, il y a une règle immuable qu'il nous faut respecter: L'élément JSX ne doit posséder qu'un seul parent. Cette contrainte est à l'origine de beaucoup de casse-têtes de par le fait qu'on préfèrerait éviter de polluer notre HTML avec des centaines de `<div>` ou de `<span>`... Heureusement pour nous, React offre la possibilité de retourner un parent invisible, un **fragment**:

```
return ( <React.Fragment> Elements dans le fragment </React.Fragment> )
```

ou en plus court:

```
return ( <> Elements dans le fragment </> )
```

Rendus basés sur une itération

Le JSX étant encore une fois un type de retour valide pour notre application, il est possible de provoquer le rendu d'X éléments de template en passant par la fonction `.map()`. Attention cependant, pour travailler, React demandera un moyen d'identifier de façon unique chaque composant. Pour cela, il nous faudra alimenter un attribut `key`:

```
return (
  <ul>
    {dogs.map(d => <li key={d.id}> {d.name} <li>)}
  </ul>
)
```

Le Style conditionnel

Pour styliser notre application de façon conditionnelle, il est possible de faire appel à des ternaires au sein de l'attribut classe ou de l'attribut style de nos balises:

- classes:

```
<div className={booleen ? 'classVrai' : 'classFaux'}> </div>
```

- styles:

```
<div style={{backgroundColor: booleen ? 'valueVrai' : 'valueFaux'}}> </div>
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les Portails

Respecter la structure HTML

Dans une application de type React, les conventions de placement de notre hiérarchie de balises sont difficilement respectées de par le fait que de base, tout va se trouver dans une balise `<div>` se trouvant dans le body de notre unique page HTML.
Pour pouvoir faire par exemple le rendu d'un modal, il nous faudrait, idéalement, provoquer son apparition dans un élément en dehors du reste de notre application. Pour ce faire, React nous offre la possibilité d'utiliser une fonction se nommant `createPortal()`.

Utilisation

La fonction createPortal() provenant de ReactDOM nous demandera deux paramètres: Un **élément JSX** (un composant) ainsi qu'un emplacement sous la forme d'une **Node** dans l'HTML (obtenu via les méthodes usuelles du Javascript):

```
render() {
  return ReactDOM.createPortal(
    <MyComponent />,
    document.querySelector("#id-destination")
  )
}
```

props.children

L'appel d'un composant par ses balises ouvrante et fermante nous offre la possibilité de placer de l'HTML à l'intérieur de notre composant de sorte à créer un composant plus générique. Pour demander, au sein du composant, le rendu du HTML injecté à partir du parent, on peut utiliser `props.children` :

```
return (
  <>
    ...
    {props.children}
    ...
  </>
)
```

Les Hooks

Qu'est-ce qu'un Hook, en fait ?

Un hook n'est en réalité ni plus ni moins qu'une fonction Javascript avec comme prefix le mot `use`.

De par l'ajout de ce préfixe, React va nous autoriser si l'on le veut à ajouter à notre fonction d'autres hooks. Via l'utilisation de ces autres hooks, il est possible de cumuler du code personnalisé et du code se servant de hooks de base, puis de retourner les variables / fonctions issues de notre code.

useRef() (1/2)

Si l'on doit réaliser dans notre application la récupération de valeurs se trouvant dans des composants HTML, mais uniquement à un moment, il serait un peu overkill d'utiliser une variable d'Etat. En effet, le re-rendu des composants à chaque changement de la valeur dans un input pourrait provoquer des fonctions en boucle alors que l'on ne le veut pas.

Pour éviter cela, il est possible d'utiliser le hook `useRef()`:

```
const myInputReference = useRef()
```

useRef() (2/2)

Pour pouvoir fixer la référence à notre élément HTML, on va utiliser l'attribut ref disponible sous tous les éléments HTML de notre template:

```
<input type="text" ref={myInputReference} />
```

La récupération de l'élément au sein d'une fonction passera ensuite par l'utilisation de l'attribut `.current`:

```
getInputValue() {
  const inputValue = myInputReference.current.value

  // Suite du code
}
```

useMemo()

Dans l'écosystème React, le re-rendu d'un composant va causer le re-calcul de toutes ses fonctions. Si certaines d'entre elles entrainent un calcul de longue durée (recherche d'un élément dans un Array), alors il se peut qu'un lag se ressente en cas de re-rendus fréquents. Pour éviter cela, il est possible d'utiliser `useMemo()` :

```
const selectedItem = useMemo(
  () => items.find(i => i.isSelected),
  [items]
  )
```

memo()

Dans le cas où l'on veut éviter le re-rendu d'un composant si ses props n'ont pas changé, on peut également au moment où l'on exporte le composant, l'entourer d'une fonction `memo()` de sorte à informer React que ce composant doit avoir un rendu évaluatif.

```
import { memo } from 'react'

const MyComponent = (props) => {
  return (
    <>
      ...
    </>
  )
}

export default memo(MyComponent)
```

Problème avec les fonctions

Dans l'écosystème React, toutes les fonctions vont être différentes, quand bien même leur code ou leurs dépendances ne changeraient pas. Dans le cadre de l'utilisation de `memo()`, cela pourrait causer un re-rendu du composant.

Pour résoudre le problème, le hook `useCallback()` nous offre la possibilité d'entourer nos fonctions par un Hook de sorte à les fixer et d'éviter X versions de la même fonction.

useCallback()

```
import { useCallback } from 'react'
import allProducts from '../data/products.js'

const MyComponent = () => {
  const MyFonction = useCallback((text) => {
    const filteredProducts = allProducts.filter(p => p.name.startsWith(text))

    setProducts(filteredProducts)
  }, [])

}
```

Attention à bien placer les dépendances dans le tableau des dépendances sous peine d'avoir des résultats non désirables.

Nos propres Hooks

Pour créer nos propres hooks, il nous suffit de créer des fonctions débutant par `use`. Via ce préfixe on se voit offrir la possibilité d'utiliser également des hooks natifs:

```
export const useDebounce = (value, delay = 500) => {
  const [debounceValue, setDebounceValue] = useState(value)

  useEffect(() => {
    const timeout = setTimeout(() => {
      setDebounceValue(value)
    }, delay)

    return () => clearTimeout(timeout)
  }, [value])

  return debounceValue
}
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Envoyer des requêtes HTTP

Fetch()

Si l'on a besoin d'accéder à des données se trouvant en dehors de notre application, il nous faudra généralement utiliser le protocole HTTP pour faire appel à des API. Dans le monde du Javascript natif, ceci va se réaliser via l'utilisation de la fonction `fetch()` qui retournera une promesse:

- A l'ancienne

```
fetch(url).then(value => { ... }).catch(error => { ... })
```

- Depuis ES6

```
try { const value = await fetch(url) } catch (error) { ... }
```

Extraire nos données

Lorsque l'on utilise la méthode native Javascript, il nous faudra également ajouter certains éléments pour obtenir le résultat escompté, tel que:

```
const response = await fetch(url, {
  method: 'POST',
  headers: {
    "Content-Type": "application/json",
    "Authorization": "Basic username password"
  },
  body: JSON.strigify(data)
})

const responseData = response.json()
```

Axios

Généralement, les requêtes HTTP seront cependant faites avec **axios** qui offre la possibilité de simplifier le parsing des éléments JSON:

```
const config = {
  headers: {
        "Content-Type": "application/json",
        "Authorization": "Basic username password"
      }
}

const { data: responseData } = await axios.post(url, data, config)
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les Props

Un enfer de props

Dans l'écosystème React, le passage d'information d'un parent vers un enfant ou dans le sens inverse passera par les props.

Cependant, dans le cadre d'une application ayant un arbre généalogique très important de composants, le passage de props va commencer à se faire sentir. Pour éviter ceci, on va utiliser un mécanisme de contexte global à nos composants qui sera disponible pour tous et modifiable par tous.

Au moment de changement de notre contexte, les composants l'utilisant vont alors recevoir une nouvelle version des données et mettre ainsi à jour l'affichage.

createContext()

Dans un premier temps, il va nous falloir créer un contexte de données. Ce contexte peut être vide, ce qui se traduirait par une valeur initiale de `null` (la valeur initiale est celle récupérée par un élément s'il appelle le contexte en dehors d'un Provider):

```
import { createContext } from 'react'

export const MyFirstContext = createContext()
```

Le Context Provider

Pour indiquer à notre application qu'elle doit être englobée d'un context, il va nous falloir utiliser le composant `.Provider` disponible dans un élément de type context:

```
const myState = useState(initValue)

return (
  <MyFirstContext.Provider value={myState}>

    ...

  </MyFirstContext.Provider>
)
```

useContext()

Pour utiliser un contexte, il faut utiliser un hook : `useContext()`. Ce hook va demander quel contexte on cherche à atteindre, ce qui ressemble à ceci:

```
import { FirstContext } from '../contexts/FirstContext.js'

const [count, setCount] = useContext(FirstContext)

return (
  <button onClick={() => setElements(count + 1)}>{count}</button>
)
```

Simplifier l'utilisation

Afin d'éviter les découpement d'éléments dans plusieurs fichiers, il est plus commun de trouver des fichier regroupant à la fois la déclaration du context, des states ainsi que le retour d'un Provider muni des valeurs pré-liées:

```
export const MyContext = createContext(null)

const MyContextProvider = (props) => {
  const MyState = useState(initValue)


  return (
    <MyContext.Provider value={MyState}>
      {props.children}
    </MyContext.Provider>
  )
}
```

useReducer()

Une autre façon de faire transiter nos props de composant à composant est l'utilisation du hook `useReducer()`. Similaire à `useState()`, le principe est ici qu'il nous faut utiliser un **state** ainsi que des **actions** provoquant les modifications de ce state de façon immuable.

Contrairement à `useState()`, `useReducer()` va nous forcer à avoir une valeur initiale à notre state.

initialState

Pour fonctionner, le hook React a besoin d'une valeur initiale. Cette valeur initiale est généralement un objet Javascript plus ou moins complexe qui contiendra les attributs relatifs à la portion de l'application que ce hook va traiter. Ainsi, on peut avoir plusieurs utilisation de `useReducer()` pour chaque section de notre application.

```
const initialState = {
  counter: 0,
  error: null
}
```

Les actions

Pour fonctionner, les éléments de notre hook vont avoir besoin d'objets respectant une certaine hierarchie d'attributs:

```
export const INCREMENT_ACTION = "counter/increment"
export const INCREMENT_BY_VALUE_ACTION = "counter/increment_by_value"
export const DECREMENT_ACTION = "counter/decrement"
export const DECREMENT_BY_VALUE_ACTION = "counter/decrement_by_value"
```

Les actions

Les constantes sont utilisées dans le but d'éviter les futures erreurs typographiques. De plus, le fait de préfixer les actions d'un thème évitera par la suite les confusions en cas d'utilisation de multiples `useReducer()`. Ainsi, l'import pourra se présenter de la sorte:

```
import * as counterActions from 'chemin/vers/actions.js'
```

La fonction réductrice

Pour traiter les actions, nous allons user d'un switch() traitant le **type** et le **payload**:

```
const reducerFunction = (state = initialState, action) => {
  const { type, payload } = action

  switch(type) {
    case counterActions.INCREMENT_ACTION:
      return {...state, counter: state.counter + 1}
    case counterActions.DECREMENT_ACTION:
      return {...state, counter: state.counter - 1}
    case counterActions.INCREMENT_BY_VALUE_ACTION:
      return {...state, counter: state.counter + payload}
    case counterActions.DECREMENT_BY_VALUE_ACTION:
      return {...state, counter: state.counter - payload}
    default:
      return {...state}
  }
}
```

Provoquer les changements

Pour initialiser le fonctionnement de notre contexte global aux composants, on va devoir utiliser le hook `useReducer()` qui va nous donner:

- Un accès au state
- Une fonction permettant d'envoyer à la fonction réductrice les actions

```
import { useReducer } from 'react'

const [state, dispatch] = useReducer(reducerFunction, initialState)
```

Modification de nos composants

Une fois le hook utilisé, au sein de nos composants, on peut procéder de la sorte pour visualiser ou modifier le state:

```
const MyComponent = (props) => {
  const [state, dispatch] = useReducer(reducerFunction, initialState)

  return (
    <p>
      <button onClick={() => dispatch({ type: counterActions.INCREMENT_ACTION})}>
        Decrement
      </button>

      {state.counter}

      <button onClick={() => dispatch({ type: counterActions.INCREMENT_ACTION})}>
        Increment
      </button>
    </p>
  )
}
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Le Routing

Single Page Application (SPA)

Dans l'ecosystème React, nous ne créons au final que des **SPA** (Single Page Applications). Ces applications consistent en une page unique envoyée par notre serveur Web et qui se voit être modifiée par le code Javascript au niveau du navigateur lui même. Le problème majeur que cela amène est l'impossibilité de récupérer une URL dans la barre supérieure de notre navigateur dans le but de partager une page spécifique ou d'en garder la trace. De plus, les boutons **Suivant** et **Précédent** se voient devenir obsolètes.

React Router DOM

Pour utiliser le routing dans React, on va passer par l'utilisation de packages supplémentaires:

```
npm install react-router-dom localforage match-sorter sort-by
```

Via l'utilisation de ces packages, il nous est par la suite possible de créer des éléments supplémentaires interceptant les requêtes et provoquant le fameux rendu conditionnel. Ces éléments sont du type **Router**, **Route**, etc...

createBrowserRouter()

La première fonction qu'il va nous falloir utiliser est disponible dans un import de `react-router-dom`. Il s'agit de `createBrowserRouter()`:

```
import { createBrowserRouter } from 'react-router-dom'

const router = createBrowserRouter([
  { path: '/', element: <HomePage />},
  { path: '/about', element: <AboutPage />},
])
```

Cette fonction permet de créer un élément de type **Router** qui servira à fixer les liens entre les routes (les urls) et les composants à rendre dans notre application.

RouterProvider

Vient ensuite l'autre import obligatoire de notre processus: `RouterProvider`. Ce composant React, fourni par le même package que le précédent, a pour rôle de fixer un emplacement du rendu des routes. C'est grâce à lui que l'on peut indiquer la nouvelle racine de notre application dans le fichier **index.js**:

```
import { RouterProvider } from 'react-router-dom'

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>,
)
```

Notre souci avec React

Dans une application React, on est, quand bien même on utilise le routing simulé par le package **react-router-dom**, dans le cadre d'une SPA. Suite à ce constat, on ne doit pas oublier le souci qu'amène le passage d'une page à l'autre en Javascript.

En réalité, si l'on demandait une nouvelle page via l'utilisation de la barre de navigation ou des balises classiques, on provoquerait l'envoi d'une requête et l'on aurait en réponse de nouveau notre sempiternelle **index.html**.

Cependant, toutes les variables seraient de nouveau initialisées comme lors de notre arrivée sur le site web.

Naviguer entre nos pages

Pour permettre une "navigation" entre nos différentes routes, il ne faut pas faire appel à des balises `<a>` classiques mais à des composants issus du package `react-router-dom`. Ces composants sont de deux types:

- Pour les liens classiques `<Link>`. Attention à bien utiliser l'attribut `to` pour y placer l'équivalent de `href`

```
<Link to="/chemin/vers/route">Texte du lien</Link>
```

- Pour les liens où l'on a besoin de savoir si l'on est actuellement sur la route `<NavLink>`

Utilisation de NavLink

`NavLink` permet d'avoir plus de contrôle sur le styling de nos liens en fonction de si la route actuelle correspond ou non à l'attribut `to`
Pour personnaliser son fonctionnement, il est possible d'utiliser une fonction:

```
const handleNavLinkClasses = ({isActive, isPending}) => {
  return isPending ? "pending-class" : isActive ? "active-class" : ""
}
```

```
<NavLink to="/chemin/vers/page" className={handleNavLinkClasses}>Texte</NavLink>
```

Gérer les erreurs

Si l'on le veut, il est possible d'informer notre routing d'une page servant, pour tel ou tel route, de page de gestion d'erreur. Via cette page, il est possible de gérer les problèmes de type **Resource Not Found - 404**. Pour cela, cela se trouve dans les objets de type `Route`, donc dans la création de nos routes:

```
import { createBrowserRouter } from 'react-router-dom'

const router = createBrowserRouter([
  { path: '/', element: <HomePage />, errorElement: <ErrorPage />}
])
```

Récupérer l'erreur dans la page

Pour obtenir l'accès à l'erreur ayant provoqué le rendu de la page d'erreur, il est possible d'utiliser un hook: `useRouteError()` qui va renvoyer l'erreur récupérée par le router. Une fois récupérée, il est possible d'en extraire des infos pour les afficher dans notre page:

```
export default function ErrorPage() {
  const error = useRouteError();
  return (
    <div id="error-page">
      <h1>Oops!</h1>
      <p>Sorry, an unexpected error has occurred.</p>
      <p>
        <i>{error.statusText || error.message}</i>
      </p>
    </div>
  );
}
```

Routes dynamiques

Imaginons désormais que l'on veuille avoir des routes pour le détail de nos produits. Ce genre de route aurait besoin d'accéder à un paramètre dans l'url correspondant au nom ou à l'identifiant de notre élément. Dans le monde de React Router, ces routes vont se traduire par des objets de ce genre:

```
const router = createBrowserRouter([
  { path: '/product/:productId', element: <ProductDetails />, errorElement: <ErrorPage />}
])
```

Ce que l'on met à droite du caractère `:` deviendra variable de route, et l'on peut ainsi par la suite récupérer `productId` au sein de notre composant. Pour ce faire, on va se servir d'un hook React prévu pour.

useParams()

Dans le cas où l'on souhaiterait accéder à nos variables de routes, il faut savoir que chacune d'entre elles se verront ajoutées en tant que propriétés dans un objet Javascript. Cet objet est récupérable via l'utilisation du hook `useParams()`.

Les attributs de cet objet auront comme nom de clé les noms de variables de root que l'on a fixé au niveau de notre routing. On peut ainsi directement utiliser le deconstructing Javascript pour en extraire les valeurs désirées:

```
const { productId } = useParams()
```

useSearchParams()

Dans une URL, il est aussi possible d'avoir des paramètres optionnels du type:

```
http://route/vers/page?paramA=valueA,valueB&paramB=value
```

Ce genre de paramètres est récupérable dans notre routing via l'utilisation d'un autre hook. De par leur côté modifiable au sein de notre page, le hook va fonctionner un peu comme **useState()** et nous retourner un getter et un setter des paramètres:

```
const [searchParams, setSearchParams] = useSearchParams();

const { paramA, paramB } = searchParams
```

Le Loader

Afin d'éviter les récupérations de variables d'identifiant puis la récupération dans nos données de l'élément à cet ID, il est possible au sein de notre routeur de récupérer directement l'élément, en amont du rendu de notre page.

Pour ce faire, on va utiliser ce que l'on appelle un **Loader** (une fonction dont l'objectif est l'envoi d'un objet plus ou moins complexe) que l'on peut ensuite utiliser dans notre page via un hook spéficique.

```
const router = createBrowserRouter([
  { path: '/product', element: <ProductList />, loader: productsLoader}
])
```

useLoaderData()

Pour récupérer par la suite les valeurs retournées par notre loader, il suffit, au sein de notre composant, d'utiliser le hook `useLoaderData()`. De par l'utilisation de ce hook, on évite, au sein de notre composant, un appel à plusieurs hooks voire des requêtes API. On peut ainsi être également sûr que les données obtenues seront les mêmes entre chaque rendu:

```
const ProductListingPage = () => {
  const products = useLoaderData()

  // Suite du composant
}
```

Un Layout commun

Pour éviter les problèmes de répétition d'éléments entre nos pages, il est possible d'utiliser ce que l'on nomme une route **Layout** (ou route racine, **Root route**). En faisant en sorte que toutes les routes passent forcément par celle-ci, il est possible de fixer des éléments communs. Au sein de cette route, il y aura le rendu de la spécificité des routes de notre application via le principe d'enfant et de parent (la route racine est ainsi le parent, et toutes les routes de notre application ses enfants). De sorte à avoir le rendu des enfants dans une route parent, il va nous falloir utiliser un second élément de routing équivalent à RouteProvider, `Outlet`.

La route racine

La route racine est au final un composant React contenant toute la structure commune de nos pages (par exemple la barre de navigation) ainsi qu'à un emplacement l'éqivalent d'un **props.children** spécifique au système de routing, l'élément `Outlet`:

```
return (
  <>
    <header>
      <nav>
        ...
      </nav>
    </header>
    <main>
      <Outlet />
    </main>
  </>
)
```

Changements dans notre routing

Cette utilisation du routing va demander l'utilisation de parents et d'enfants, que l'on peut spécifier au sein de nos routes en paramètre de la fonction `createBrowserRouting()`:

```
import { createBrowserRouter } from 'react-router-dom'

const router = createBrowserRouter([
  { path: '/', element: <Layout />, errorElement: <ErrorPage />, children: [
    { path: '/', element: <HomePage />},
    { path: '/about', element: <AboutPage />}
  ]},
])
```

useNavigate()

Si l'on souhaite naviguer, non pas en cliquant sur un lien, mais en suite d'une execution de code métier, il va nous falloir avoir accès à la capacité de provoquer une navigation au niveau de notre code Javascript. Pour cela, il est possible d'exploiter le hook `useNavigate()` qui nous donne l'accès à une fonction pouvant prendre en paramètre une chaine de caractère correspondant à la route à atteindre:

```
const MyComponent = () => {
  const navigate = useNavigate()

  const handleClick = () => {
    console.log("Je navigue dans mon site...")
    navigate("/route/a/atteindre")
  }

  return <button onClick={handleClick}>Click me to navigate</button>
}
```

Routes sécurisées (1/2)

Grâce à cela, il est possible de créer ce que l'on appelle des routes sécurisées. Ces routes ne sont au final ni plus ni moins que des composants React permettant un rendu de l'élément `props.children` en cas de vérification d'une variable précise:

```
const  SecuredRoute = (props) => {
  const navigate = useNavigate()
  const { user } = useContext(AuthContext)

  if (user) {
    return <>{props.children}</>
  } else {
    navigate("/auth/login")
  }
}
```

Routes sécurisées (2/2)

Une autre façon de faire des routes sécurisées est d'utiliser le composant de redirection disponible dans le package. Ce composant, en cas de rendu par React, va provoquer la redirection. De la sorte, on peut, au lieu de faire appel au hook, simplement retourner le composant:

```
const  SecuredRoute = (props) => {
  const { user } = useContext(AuthContext)

  return <>
    {user ? props.children : <Redirect to="/chemin/vers/login" />}
  </>
}
```

Loader avec paramètres

Si l'on souhaite faire un loader exploitant des paramètres de routes, il faut savoir que par défaut, React router DOM va envoyer à notre fonction de loader un objet dont on peut extraire les paramètres:

```
export async function loader(object) {
  const { params } = object
  const contact = await getContact(params.contactId);
  return contact;
}
```

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Store Redux

Qu'est-ce que Redux ?

Redux est un package supplémentaire permettant de grandement faciliter la gestion de l'état de notre application. Similaire au hook `useReducer()` disponible nativement dans React, Redux peut cependant permettre la gestion de plusieurs fonctions réductrices et de plusieurs stores sans aucun problème. De plus, il est capable de gérer des **side effects**, des fonctions (synchrones et asynchrones) provoquées par l'envoi d'un dispatch dans notre store. Pour ajouter Redux à notre application, on va passer par `npm`:

```
npm install redux @reduxjs/toolkit
```

Création du Store

Pour créer notre store, il nous faut une fonction réductrice ainsi qu'un état initial. Une fois fait, on utilise `createStore()`:

```
const initialState = {
  value: 0
}

const counterReducer = (state = initialState, action) {
  const { type } = action
  switch (type) {
    case 'counter/incremented':
      return { value: state.value + 1 }
    case 'counter/decremented':
      return { value: state.value - 1 }
    default:
      return state
  }
}

const store = createStore(counterReducer)
```

@reduxjs/toolkit

Si l'on continue via l'utilisation classique de Redux, on va vite arriver à un casse-tête en cas de modification d'objets complexes. En effet, tout comme le hook `useReducer()` et toute fonction réductrice, il nous faudra appliquer le principe d'immutabilité pour faire fonctionner l'application. En retournant à chaque fois un nouvel objet de State, React va provoquer le rendu de tous les éléments en dépendant. Néanmoins, repasser à chaque fois toute une hierarchie d'attributs alors que l'on veut n'en modifier qu'un risque de poser des soucis de lisibilité. C'est pour cela que l'on va utiliser le package **@reduxjs/toolkit** !

Terminologie

- Une **tranche** (slice) est un morceau de notre store servant à centraliser et à traiter les différentes données.
- Une **action** est un élément envoyé à notre store, relayé à la tranche concernée, contenant le nom de la fonction réductrice à appeller ainsi que d'eventuels paramètres.
- Le **payload** est un objet plus ou moins complexe contenant les paramètres de la fonction réductrice
- le **store** est le rassemblement de toutes les tranches ainsi que le lieu de passage de toutes les actions

Création d'une tranche

Ce package va, en plus de simplifier la création des actions et des tranches de notre contexte, permettre le passage de modification de façon mutable. Ces modification seront analysées par la librairie et rendues immutables de sorte à provoquer le changement de state:

```
const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    incremented: state => { state.value += 1 },
    decremented: state => { state.value -= 1 }
  }
})
```

Les Actions et le Store

Une fois la tranche réalisée, il est possible d'en extraire les actions. De par le passage d'un attribut `name` au moment de la création de la tranche, les actions auront automatiquement comme valeur en `type` le nom de la tranche suivi du nom de l'action.

```
export const { incremented, decremented } = counterSlice.actions
```

Pour créer le store, il nous suffit d'obtenir le reducer de notre tranche avant de le passer aux reducers de notre store:

```
const store = configureStore({
  reducer: counterSlice.reducer
})
```

Plusieurs tranches

En cas d'utilisation dans notre application de plusieurs tranches (dans un souci de ne pas transmettre les informations spécifiques à une section de l'application à une autre), on va devoir changer légèrement la syntaxe de la création de notre store:

```
export default configureStore({
  reducer: {
    users: usersReducer,
    posts: postsReducer,
    comments: commentsReducer
  }
})
```

useSelector()

Dans le cas de l'utilisation d'un contexte Redux, il nous faudra bien entendu pouvoir récupérer les éléments se trouvant dans telle ou telle tranche de notre application. Pour cela, il est possible, au moyen d'un hook, de passer une fonction fléchée précisant l'objet (la tranche de données) que l'on souhaite récupérer. Ce hook porte le doux nom de `useSelector()` et se présente comme ceci:

```
export default configureStore({
  reducer: { productsSlice: productsReducer }
})
```

```
const products = useSelector(state => state.productsSlice.products)
```

useDispatch()

Dans le but de provoquer nos actions, il nous faut maintenant utiliser un autre hook permettant l'envoi d'actions (objets constitués d'un type et d'un payload) à notre store. Si l'on a utilisé le package **@reduxjs/toolkit**, les actions sont prévues comme ayant une syntaxe évitant les déclenchements de plusieurs modifications de state dans le store. Dans le cas contraire, attention à la valeur de **type**:

```
const dispatch = useDispatch()

const addProduct = (product) => {
  dispatch(actions.addProduct(product))
}
```

Les Thunks

Si l'on souhaite réaliser de la logique asynchrone, il va nous falloir placer cela dans des fonctions à part. Ces fonctions doivent, idéalement, provoquer une, deux ou trois actions:

- **Pending** permettant d'indiquer l'attente d'une réponse
- **Success** permettant d'ajouter la réponse au contexte
- **Failed** permettant d'ajouter l'erreur au contexte

Notre fonction asynchrone va donc devoir, en son corps, traiter le dispatching de trois potentielles actions !

Syntaxe

```
const fetchProductById = productId => {
  return async dispatch => {
    dispatch(getProductPending())

    try {
      const product = await productAPI.getProductById(productId)

      dispatch(getProductSuccess(product))
    } catch (err) {
      dispatch(getProductFailure(err))
    }
  }
}
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Déploiement

Site statique

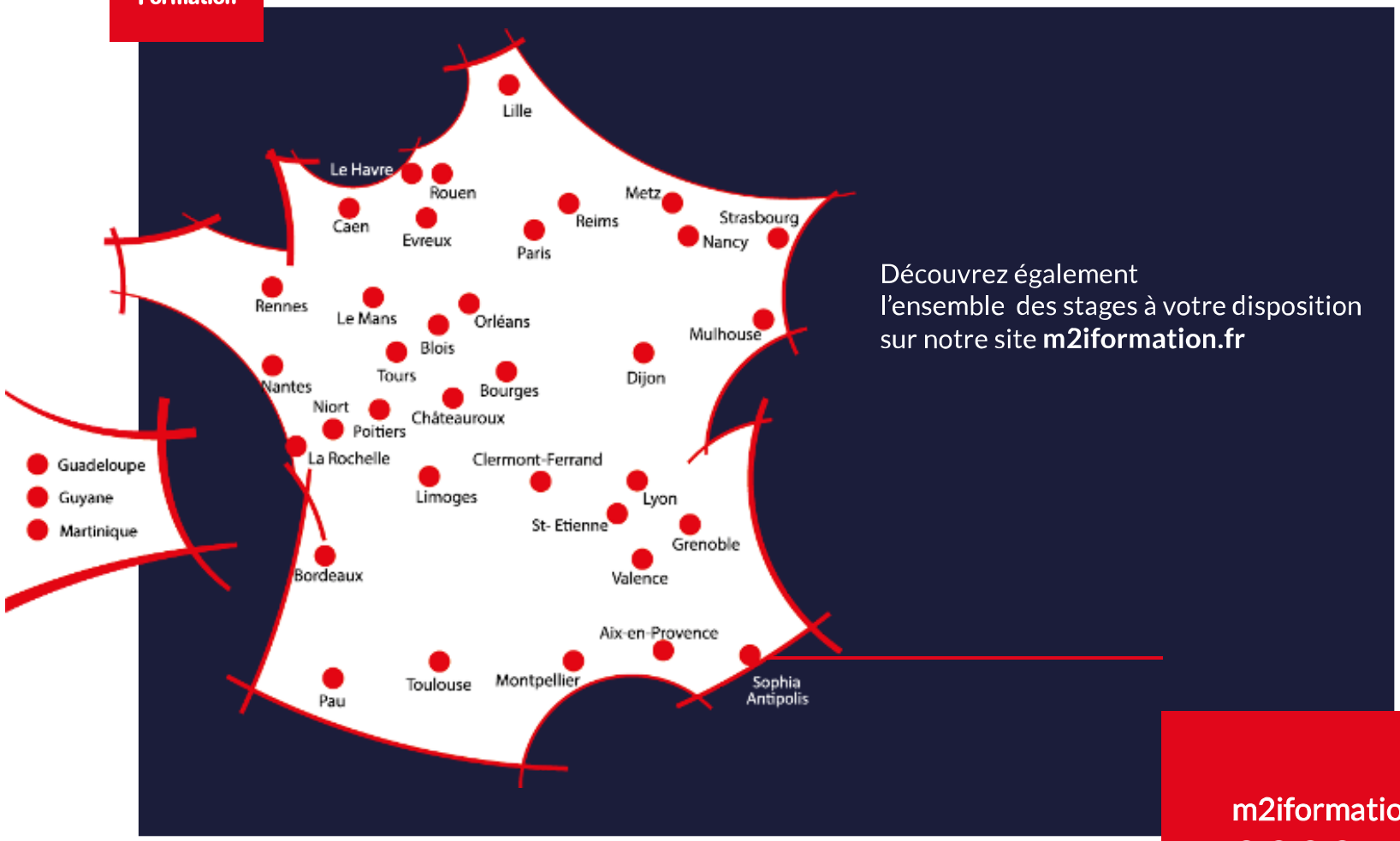
Pour créer une version production de notre application, il va falloir utiliser la commande:

```
npm run build
```

Cette commande va créer un dossier **dist** qui va contenir la version statique de notre application React. Ce rendu est compatible avec la majorité des serveur Web actuels.

NGINX

Pour avoir facilement un serveur Web permettant de faire tourner notre application React, le moyen le plus simple est de faire appel à **NGINX**, qui est un serveur web (entre autres choses). Dans sa configuration par défaut, NGINX va publier par HTTP au port 80 le contenu du dossier se trouvant, sur la version Linux, dans `/usr/share/nginx/html`. Une solution "simple" consiste alors à créer un conteneur **Docker** de l'image NGINX se chargeant de synchroniser le dossier `/usr/share/nginx/html` du conteneur et le dossier `/dist` de notre build.



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr